The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study

Martin Feilkas and Daniel Ratiu and Elmar Jürgens
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
feilkas|ratiu|juergens@in.tum.de

Abstract

Architecture defines the components of a system and their dependencies. The knowledge about how the architecture is intended to be implemented is essential to keep the system structure coherent and thereby comprehensible. In practice, this architectural knowledge is explicitly formulated only in the documentation (if at all), which usually gets outdated very soon. This leads to a growing amount of implicit knowledge during evolution that is especially volatile in projects with high developer fluctuation.

In this paper we present a case study about the loss of architectural knowledge in three industrial projects by tackling the following research questions: 1) to what degree is the architectural documentation kept consistent with the code? 2) how well does the documentation reflect the intended architecture?, 3) how big is the architectural decay?, and 4) what are the causes for inconsistencies? We answer these questions by investigating the architecture documentation, the source code, and by performing interviews with developers.

The most important outcomes of our study are: the informal documentation and the source code are not kept consistent with each other, none of them completely reflects the intended architecture, and even developers taken individually are not completely aware of the intended architecture. Quantitatively, between 70% and 90% of these inconsistencies are caused by flaws in the documentation and between 10% and 30% represent architectural violations in the code.

1 Introduction

The architecture defines the structure of a software system in terms of components and (allowed) dependencies. A suitable architecture is a fundamental prerequisite for having evolvable and understandable systems [5]. Developers need knowledge about the intended architecture of a system whenever they do any modification. Without this knowledge programmers can break the architectural integrity of the system accidentally, even by making only small code changes.

The widely used programming languages today have very primitive mechanisms for making the architecture in the code explicit. Therefore, in everyday industrial practice, the information about the architecture is contained in external documentation in form of diagrams and natural language texts that often originate from early phases in the design of the system. During system evolution, the architecture often needs to be adapted, extended and modified in case of changes to the requirements, additional features or simply by new insights about shortcomings of the initial design. These changes are inevitable even if an 'optimal design strategy' is used [13], needless to say that this effect is amplified in an industrial environment. When these changes to the intended architecture happen, they are often (unintentionally) not introduced into the architecture documentation and not propagated to other team members [10]. This leads to a gap between the intended architecture of the system, how different developers perceive it, how it is made explicit in the documentation and how it is actually implemented in the code.

In Figure 1-left illustrates intuitively the ideal situation when all developers possess accurate knowledge about the system's architecture, the architecture is accurately documented and accurately implemented in the code. The right side of the figure illustrates the situation typically encountered in industrial projects: Different developers understand the architecture of big systems in (slightly) different manners, with none of them having an accurate view of the intended architecture. Furthermore, only a part of the intended architecture is documented and only a part of the code complies with it. As depicted in Figure 1-right, the *loss of architectural knowledge* can be observed in different forms: missing architectural information in the documen-

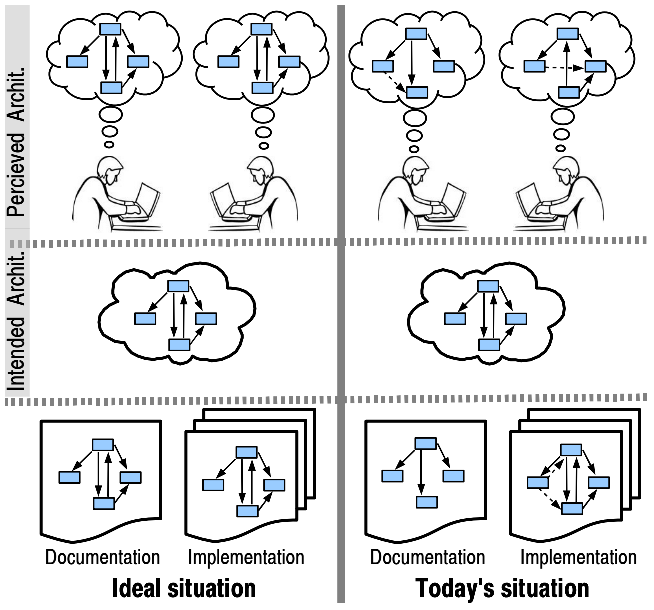


Figure 1. Loss of architectural knowledge

tation, violations of the architecture in the code, problems in keeping the code and documentation synchronized and different perceptions of the intended architecture by the developers.

In this paper we present an empirical case study on the loss of architectural information within the evolution of industrial software. We quantify in what degree the documentation and the code are kept consistent. Furthermore, by interviewing the developers, we evaluate whether the differences are violations of the architecture in the implementation or if they are insufficiencies of the documentation. This case study has been done on three industrial business systems of different age and functionality.

Outline. In Section 2 we briefly describe our approach for analyzing the consistency between the documented architecture and the implementation. In Section 3 we present three case studies we performed in a collaboration with MunichRE. In Section 4 we discuss the lessons that we learnt following our experience. Section 5 poses threats to validity that could influence our conclusions. We end this paper with presenting related work in Section 6 and conclusions in Section 7.

2 Technique and methodology for architecture analysis

In this section our approach to describe the architecture in a machine-readable form is presented and our technique to analyze the conformance between the documented architecture and the code is explained. We exemplify our approach on C# although the technique can be generalized easily.

2.1 Architecture conformance analysis

Describing the architecture. In order to do an automatic analysis the architecture is specified in terms of a set of hierarchical components $comp$ and policies pol among them. So an architecture description $arch$ can be formalized as

$$arch = (comp, pol).$$

Components are the main structuring entities in this description mechanism. The hierarchy is expressed as a predicate

$$isSubComp : comp \times comp \rightarrow bool.$$

Policies can also be regarded as a predicate

$$pol : comp \times comp \rightarrow bool.$$

A component is always allowed to access its subcomponents:

$$isSubComp(c_1, c_2) \Rightarrow pol(c_1, c_2)$$

If there is a policy defined between two components, this specifies that in the implementation the elements that correspond to these components may have dependencies between each other (in the specified direction). Every dependency that is not explicitly allowed by such a policy is forbidden. In many cases the architecture specifies a structure of the system such that certain components are decoupled. If no policy explicitly allows two components to be coupled with each other, then no dependency is allowed at the code level.

Describing the code. In object-oriented systems every element of the code is encapsulated in types (in .NET these types are defined as classes, structs, enums, ...). So for architecture conformance analysis a system

$$sys = (types, dep)$$

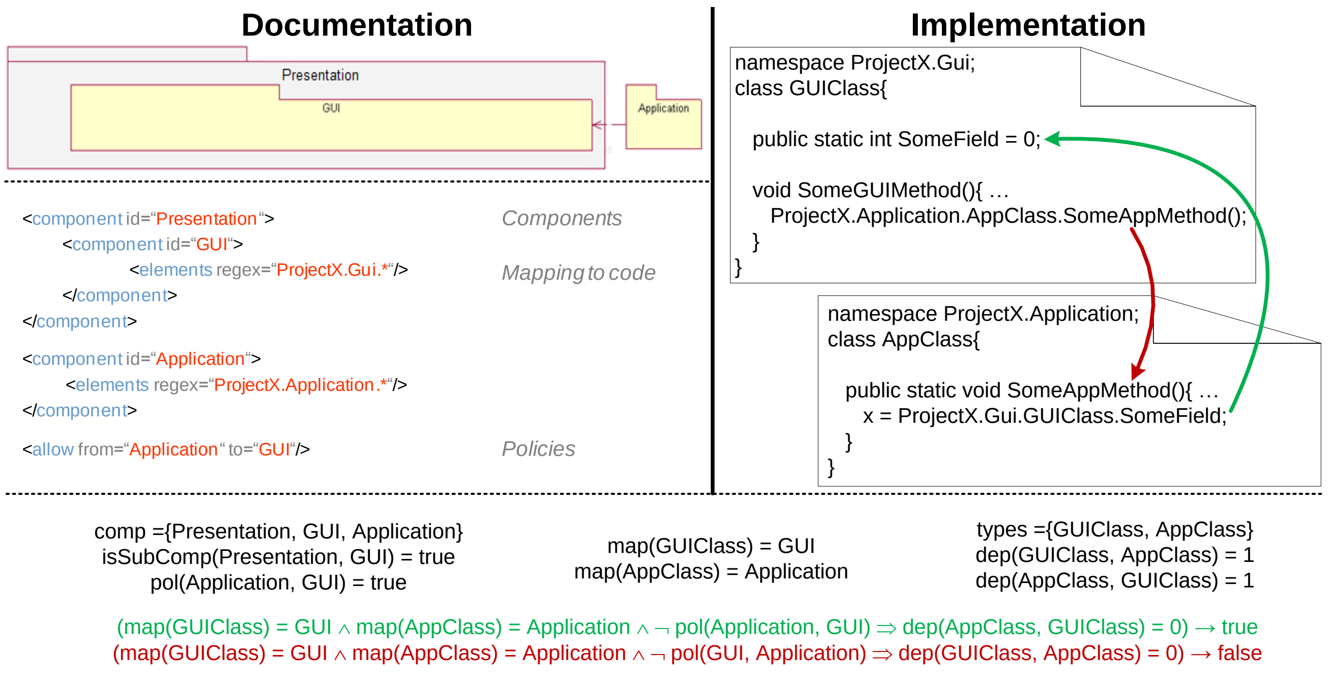
can be regarded as a set of types $types$ and a set of dependencies dep between them. The number of *dependencies* is expressed by the function

$$dep : types \times types \rightarrow \mathbb{N}$$

A type t_1 is dependent on another type t_2 if t_2 (or one of its elements) is used in t_1 as defined in Table 1.

Invocation of a method/constructor
Access of a property or field
Extending a class/struct, implementing an interface
Usage of a class/struct/enumeration as a type (for a field, variable or parameter)
Annotation of an attribute

Table 1. The dependencies in the code



```
<component id="Presentation">
    <component id="GUI">
        <elements regex="ProjectX.Gui.*"/>
    </component>
</component>
<component id="Application">
    <elements regex="ProjectX.Application.*"/>
</component>
<allow from="Application" to="GUI"/>
```

```
namespace ProjectX.Gui;
class GUIClass{

  public static int SomeField = 0;

  void SomeGUIMethod(){ …
    ProjectX.Application.AppClass.SomeAppMethod();
  }
}
```

```
namespace ProjectX.Application;
class AppClass{

  public static void SomeAppMethod(){ …
    x = ProjectX.Gui.GUIClass.SomeField;
  }
}
```

comp ={Presentation, GUI, Application}
isSubComp(Presentation, GUI) = true
pol(Application, GUI) = true

map(GUIClass) = GUI
map(AppClass) = Application

types ={GUIClass, AppClass}
dep(GUIClass, AppClass) = 1
dep(AppClass, GUIClass) = 1

$(map(GUIClass) = GUI \wedge map(AppClass) = Application \wedge \neg pol(Application, GUI) \Rightarrow dep(AppClass, GUIClass) = 0) \rightarrow true$

$(map(GUIClass) = GUI \wedge map(AppClass) = Application \wedge \neg pol(GUI, Application) \Rightarrow dep(GUIClass, AppClass) = 0) \rightarrow false$

Figure 2. The architecture description mechanism

Checking conformance. To map the architecture description to the system we need to define a *code mapping* as a function $map : types \longrightarrow comp$. If the architecture description is completely in conformance with the implementation (no not allowed dependencies) the following condition must hold:

$$map(t_1) = c_1 \wedge map(t_2) = c_2 \wedge \neg pol(c_1, c_2) \Rightarrow dep(t_1, t_2) = 0$$

We will write 'there are x differences between from component c_1 to c_2' if the condition does not hold with the left hand side being true but the right hand side being false with $dep(t_1, t_2) = x$ and $x > 0$.

Technical execution of the analysis. The architecture is specified in a machine-readable form in a XML-file. Figure 2-left shows an example architecture: it contains three components (i.e. Presentation, GUI and Application), GUI is a subcomponent of Presentation and there is a policy defined that allows the Application component to be dependent on GUI. The figure illustrates how this simple example is described using XML. This file contains a simple XML description of hierarchical components and their mapping to types in the source code based on regular expressions. These expressions are used to map the full-qualified names of the types in the implementation to the components. Additionally, allow-tags are defined to describe the policies of how the components may depend on each other. The right hand side of Figure 2 illustrates the implementation level: A green arrow represents an allowed dependency from the Application to the GUI component. Using a red arrow a dependency is shown that violates the specification on the left hand side due to an access from the GUI to the Application component. The formalisation is given in the lower part of the figure.

All projects that are subject to the case study are implemented using the .NET framework. The analysis is performed using the Continuous Quality Assessment Toolkit (ConQAT) [2][1]. It calculates the set of dependencies that are not explicitly allowed in the architecture description.

2.2 Methodology

Analysis steps. Figure 3 illustrates the steps needed to perform an architecture analysis on a system:

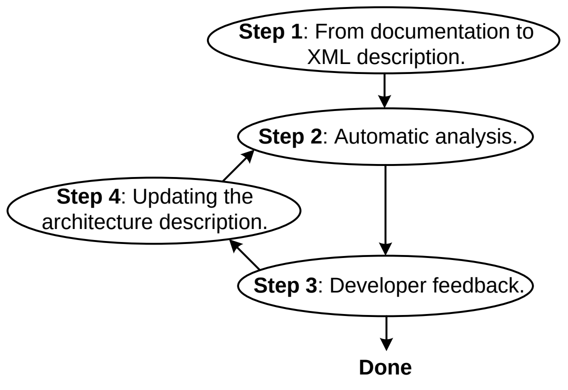

Step 1: From documentation to XML description.
Step 2: Automatic analysis.
Step 4: Updating the architecture description.
Step 3: Developer feedback.
Done

Figure 3. Analysis steps

[1] `conqat.in.tum.de`

Step 1: Translation of the architecture documentation into the machine-readable XML representation. In this step we obtain a XML description of the architecture as it was originally documented.

Step 2: Checking the conformity of the code with respect to the current XML architecture description. This step uses the automatic analysis and has a list of inconsistencies between the XML document and the code as a result. These inconsistencies are either due to an insufficient description of the intended architecture in the current XML-file or violations of the intended architecture in the code.

Step 3: Discussion of the results with the developers. We discussed the results of step 2 with the developers in order to classify the differences into violations of the intended architecture at code level or deficiencies in the current XML architectural description (this classification uses the implicit knowledge about the system that is only in the heads of developers, if at all). If there are differences in the output of the analysis that are not regarded as violations, the XML description of the architecture does not represent the intended architecture yet. In this case the architecture description has to be adapted by doing step 4. If the developers regard all of the differences as code deficiencies then the process is complete.

Step 4: Refinement of the architecture specification by considering the implicit knowledge that was not present in the documentation and revealed in step 3. After that step 2 has to be performed.

After each iteration the architecture defined in the XML description converges to the intended architecture. Every modification that is necessary in the XML description (step 4) during our iterations are regarded as flaws in the original documentation due to the changes in the architecture that were not documented (due to the *architectural drift* [11]). After two to four iterations of the steps 2, 3 and 4, the architecture description was regarded as a precise specification of the intended architecture by the team members. Using the final XML description that contains the intended architecture, we are able to perform a final architecture analysis to measure the violations in the code of the intended architecture (and thereby to measure the *architectural decay* [11] of the code).

Outputs of the analysis. During the analysis process we compute the following sets:

- *missingComp*: The set of components that are implemented in the system but are missing in the documentation.

- *relocComp*: The set of components that changed their super components. A component x is called a 'relocated component' if $isSubComp(x, a)$ is specified in the documentation and $isSubComp(x, b)$ reflects the intended architecture ($a \neq b$).

- *polMod*: The set of policies that were introduced or modified during the process of the analysis (in step 4).

- dep_{all}: The set of dependencies between the components in the system:

$$|dep_{all}| = \sum_{t_1 \in types} \sum_{t_2 \in types} dep(t_1, t_2)$$

- diff_{doc}: The subset of dependencies ($\mathit{diff}_{doc} \subset dep_{all}$) that represent differences between the original documentation of the architecture and the implementation. This set is computed after the first run of step 2. These differences reflect the divergence between the documented and the implemented architecture.

- diff_{intend}: The subset of diff_{doc} that the architecture analysis revealed after the whole process was finished. The architecture description obtained after the iterations reflects the intended architecture. Therefore, all dependencies in diff_{intend} are violations in the code.

3 The case study

We start this section with presenting the research questions addressed by our case study, continue with describing the experimental setup, present the quantitative results of our analyses, and finally present an interpretation of the results measured.

3.1 Research questions

Q 1: To what degree is architecture documentation kept consistent with the implementation during system evolution? The entry point to our analyzes deals with the relation between the documented and the implemented architecture. If differences can be identified, this indicates that either the implementation violates the intended architecture or that modifications of the architecture during the system evolution are not propagated to the documentation. We answer this question by calculating the amount of differences relative to the dependencies in the code:

$$\mathit{reldiff} = \frac{|\mathit{diff}_{doc}|}{|dep_{all}|}.$$

Q 2: How well does the documentation of the architecture reflect the intended architecture? A vague or outdated description of the architecture is inadequate for conserving architectural knowledge for software maintenance and evolution. The quality of the documentation is related to the explicit knowledge about the project that the team members can always refer to. If no precise and up-to-date documentation of the architecture exists, new project members will have difficulties in learning the architecture. We measure the amount of implicit knowledge in terms of documentation flaws defined as:

$$\mathit{docflaw} = \frac{|\mathit{diff}_{doc}| - |\mathit{diff}_{intend}|}{|\mathit{diff}_{doc}|}.$$

Additionally, the numbers of components that were undocumented $|missingComp|$ or relocated $|relocComp|$ as well as the policies that had to be modified $|polMod|$ are metrics for measuring the divergence between the documented and the intended architecture.

Q 3: How big is the architectural decay? This question should clarify to what degree violations of the intended architecture can be found in the code. The architectural decay can be measured in terms of violations:

$$viol = \frac{|diff_{intend}|}{|diff_{doc}|}$$

Q 4: What are the causes of inconsistencies between the intended architecture and the code? We investigate the causes of the introduction of architectural violations into the code as well as why documentation is not kept up to date. This is a qualitative question that we answer based on the interviews with developers (step 3).

3.2 Experimental setup

Industrial environment. The case study was done in a collaboration between MunichRE and Technische Universität München. MunichRE is a big reinsurance company with about 39.000 employees worldwide. For the insurance branch it is typical to make heavy use of individually developed software to support the business processes such as sales, risk calculation or capital investments. At MunichRE software development is done mainly by external developers. This leads to relatively high fluctuation of developers. A specially tailored RUP-like engineering process has to be applied to every software project. This process prescribes that an architecture documentation must be created in every project. To ensure a seamless hand-over between developers, the maintainability of the software products must be high. Thus, the systems must be implemented in a comprehensible and homogeneous way. A prerequisite for these desiderates is to manage the architectural knowledge by making it explicit.

The projects. We investigated three typical business systems implemented in C#. We emphasize on the fact that at MunichRE these projects are considered to be of good quality and successful. They are in productive use by 10 to 150 users in different departments of MunichRE. All of these systems have been developed by developers from different software development contractors. During the initial development there were up to 12 developers involved in each project. After these systems went productive, the number of developers was reduced. The systems are all still further extended and evolved.

Project A is a typical rich client application is further developed and maintained. The system is in production for about 5 years. It provides insurance risk calculation functionalities. Currently 5 developers evolve the system. There have been personal fluctuations so that there is currently none of the initial developers in the team. The architecture documentation of Project A is a text document that contains a component diagram consisting of hierarchical components (boxes). These components are connected via arrows that represented allowed dependencies. This diagram was the most important source of architecture information.

Project B is a web-based information system. With a lifetime of 6 years in production this is the oldest of the investigated projects. It is used for doing financing and investment calculations. Currently there are 4 developers involved in maintaining the project. Like in Project A, there have been personnel fluctuations so that none of the initial developers works in the project anymore. In Project B, the architecture is described by diagrams similar to UML package diagrams.

Project C is also a web-based system. It is the youngest of the systems and under development for about 2 years. It provides functionality for managing risk information about certain clients of MunichRE. An average number of 3 developers evolve the system. It is in productive use only for about 7 months. The architecture documentation of Project C is also a text document which contains diagrams that illustrate components and their relations by boxes and arrows.

	Age	**kLOC**	**Developers**
Project A	5	454	5
Project B	6	317	4
Project C	2	495	3

Table 2. Data concerning the projects

In Table 2 we present the projects at a glance: their age, their size code and the current number of developers that maintain the systems. We remark that the size of the projects is in the same magnitude. However, Project C is the biggest even if it was the most recently started and even if it has the least number of developers assigned for maintenance and evolution.

Table 3 shows the size of the documented architecture in terms of the number of components and the number of policies (allowed accesses between two components). We should also remark the big number of policies in the case of Project A and C (they contain only 24, respectively 37, components and have allowed many policies). In contrast to projects A and C, Project B contains much more components (60) and almost the same number of policies (106) as Project C. So the description of Project B is of much finer granularity and its architecture is much more restrictive.

	Components	Policies
Project A	24	79
Project B	60	106
Project C	37	102

Table 3. Initial description of the architecture

3.3 Quantitative Results

Table 4 presents the number of modifications of the architecture description during the iterative refinement. The first column shows the number of relocated components *relocComp*. Such a relocation is, for example, moving a subcomponent of the Business component to the DataAccess component. The second column represents the number of components that were not documented and introduced during the analysis process (step 4). The third column shows how many policies have been modified or introduced additionally to the documented ones. No policies have been removed without substitution. This shows that the documented architecture usually has less policies than the architecture actually implemented. In other words, the documented architecture seems more modular than the one that is really implemented.

	relocComp	*missingComp*	*polMod*
Project A	0	2	8
Project B	6	2	24
Project C	2	1	9

Table 4. Modifications of the architecture specification during steps 3 and 4

Table 5 shows the main results of our analyses. The *dep* column presents the number of dependencies that the architecture conformance analysis identified in the systems. These dependencies were validated against the XML architecture description. The $\textit{diff}_{doc}$ column presents the results of the conformance analysis, namely the number of dependencies that did not conform to the specified architecture. The *docflaw* and *viol* the columns contain the percentage of the non-conform dependencies that are flaws in the documentation and respectively violations of the architecture in the implementation.

	dep	$\textit{diff}_{doc}$	*docflaw*	*viol*
Project A	8.254	994 (12%)	90%	10%
Project B	4.385	376 (9%)	72%	28%
Project C	5.388	(2238) 1039 (19%)	88%	12%

Table 5. Results of the analysis

System A. In system A a subset of 994 (out of 8.254) dependencies were identified as differences between the documented and the implemented architecture (12%). As Table 4 shows, 8 policies had to be added and 2 components were introduced in order to get from the documented to the intended architecture during the analysis process. After the completion of the analysis, about 10% of these differences were identified as violations of the intended architecture within the implementation. The other dependencies were mostly undocumented modifications of the architecture during the evolution of the system.

Figure 4 shows a visualization of the results of the analysis of system A. This kind of visualization, created using the graph visualization tool 'dot' [7], was integrated into the view of results in the projects' dashboards and represented the entry point in obtaining feedback from the developers. Red arrows represent violations of the specified architecture, green ones are allowed dependencies. Additionally to these visualization, detailed lists of the identified differences are used as inputs for discussion.

System B. The results of the analysis of Project B revealed a lower number of differences between the documented and the implemented architecture (376 out of 4.385). Much more adaptations of the architecture description were needed in Project B (Table 4) after the discussions with the developers to get a description of the intended architecture. The architecture description of this project was much more detailed and fine-grained in the documentation (Table 3). A relatively high share of architecture violations (28%) of the 376 differences were identified.

System C. After the first analysis of system C based on the initial architecture description we identified a very large number of differences (2.238 – effectively 42%). By looking at the system more closely it revealed that this extreme result has been affected by a single cause: The system was built using so called "data binding technique" offered by the .NET framework. This technique implies dependencies directly from GUI-parts to data access components. These dependencies were not allowed in the initial specification of the architecture. Although these dependencies are differences between the specification and the implementation of the architecture, these have in contrast to usual architecture violations not been introduced due to a developer's lack of knowledge about the architecture. For that reason the XML architecture description was modified accordingly (i.e. we introduced a new policy to allow these dependencies), and a lower value of 1039 (ca. 19%) differences was measured. This value was used for the further steps of the case study. Based on these differences 12% revealed to be architecture violations which is a very similar value to Project A. Also the number of adaptations of the documented architecture were in a similar range than in Project A (Table 4).

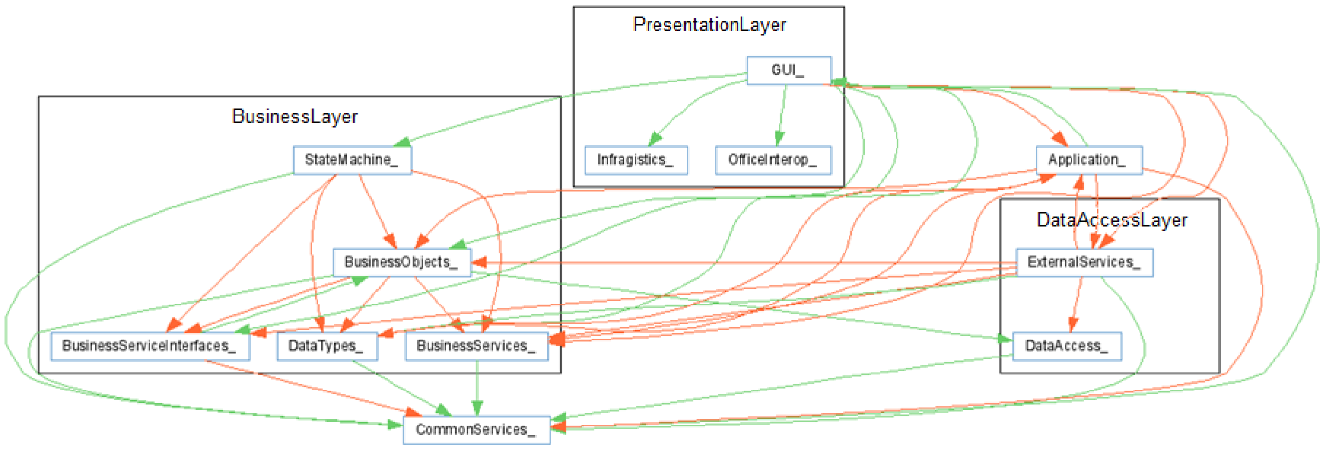


Figure 4. An example for the visualization of the results of the architecture analysis (Project A)

3.4 Interpretation of the results and answers to the research questions

Q 1: *To what degree is architecture documentation kept consistent with the implementation during system evolution?* Following our automatic analysis of the architecture conformance, a significant number of differences between the documented architecture and the implementation have been found in all three projects. As shown in Table 5 between 9% and 19% of all the dependencies that are implemented in the systems could not be identified in its corresponding specification. These differences represent either documentation flaws or violations of the intended architecture in the code. The *reldiff* values of the projects reflect that Project B has the least differences, but even in this case the amount of discrepancies is significant (every tenth dependency in the system cannot be found in the documentation). In the worst case (Project C) almost one fifth of the dependencies do not conform with the documentation. This significant desynchronization between the documentation and the code led developers regard the documentation as an unreliable source of architectural information.

Q 2: *How well does the documentation of the architecture reflect the intentions of the architect?* The documentation flaws were identified after performing interviews with the developers (step 3). Many of these interviews caused vivid discussions among developers due to their different perceptions of the intended architecture. The outcome of these interviews were policies that had to be added to the architecture specification. Table 5 shows that between 72% and 90% of the differences between the documented and the implemented architectures are due to flaws in the documentation. Table 4 summarizes the modifications of the initial documentation that were made during the analysis process. As the projects A and C needed an almost equal amount of modifications, Project B caused more changes that had to be integrated into the architecture description. The reason therefore is the more fine-grained architecture definition in Project B.

In summary most of the differences ($diff_{doc}$) must be regarded as deficiencies of the documentation. The documentation flaws are a measure of architectural drift, namely the measure in which the initially intended architecture developed further over time.

Q 3: *How big is the architectural decay?* The last column of Table 5 shows the architectural violations measured in the projects. The relative amount of architecture violations in the systems is between 10% and 28%. Although Project B has the highest relative value in that table (28%), the absolute number of violations that had to be removed in the analyzed systems is with about 100 dependencies almost equal. The fine-grained architecture description of Project B provokes that the analysis can be regarded to be more precise. Due to that more violations could be identified. Additionally, the architecture of system B is much more restrictive than the architecture description of the other projects (Table 3). As a consequence fewer dependencies are allowed an though developers more likely introduce violations.

Q 4: *What are the causes of inconsistencies between the intended architecture and the code?* Most of the documentation flaws are caused by new insights during the implementation phase. For example, system A should contain no dependencies between the GUI and the DataAccess component. However, our analysis revealed dependencies between these components. After further investigations the developers told us that these were uncritical and allowed because they were needed to access specific data during system startup. At startup time the business components that are usually used for acquiring data are not present. This is an example for a refinement of the architecture that was performed during the implementation of the system. During the design phase it was overlooked that such a dependency is necessary. However, this knowledge had not been introduced into the documentation.

The developers regard the architecture documentation as a relict from the very beginning of the project when the architecture was initially created. So the architecture documentation is seen rather as an artifact that should ease the constructive phase (design) rather than a description of the system for long-term maintenance and evolution. The relatively high amount of documentation flaws is not due to 'laziness' of the developers. Many times developers do simply not remember that there are parts in the documentation that should be modified consistently. This can be regarded as a lack of availability of ordinary documentation. Thus, documentation becomes a dead artifact that is used very infrequently. Many times the developers (especially in Project C) knew that the documentation was outdated.They explained that redocumentation activities are often postponed due to the higher priority of the implementation of new features or bug-fixes in their daily work.

During our analyzes we identified several even critical architecture violations. In Project A, for example, the following violations have been identified: several invocations of methods had been implemented that launched transactions in an non-adequate way (using the wrong interface). This caused several unneeded transactions and a significant loss of performance. This violation was caused by a developer that did not use the interfaces that were intended to be used for that purpose. The explanation of the developers was that the implementer of these pieces of code did not know which components should be used to achieve these tasks. This shows that not all of the developers were aware of the intent and the adequate usage of the architecture.

Another example for a typical reason of a violation is copy and paste programming. Many times the headers of files have been copied and pasted to be used as a template for the implementation of new classes. Unfortunately, the namespace declaration was often part of the copied lines and it had been missed to modify it accordingly. However, although that seems not so critical, it is difficult for another developer to identify this as a copy&paste problem and to understand that the class should be declared somewhere else.

4 Lessons learnt

Architectural knowledge gets lost. Developers do not understand the complete architecture of the system and especially how it is reflected in the code. The main cause is the myriad of details at the code level and the big abstraction gap between architectural specifications and the implementation. Instead of performing tedious work for understanding and recovering (guessing) the intended architecture, we advocate to an approach to conserve the architectural knowledge and how it is implemented in the code.

Conformance checking is a catalyzer for discussions. The case study revealed that a more structured documentation in a machine-readable form and an automatic analysis gives the architecture a bigger awareness in the development team. Several discussions on the correct usage of the architecture were raised during the case study. So the different views on the intended architecture by the team members were synchronized and introduced into an explicit documentation.

Continuous architecture analysis. To ensure that the architecture description will be kept consistent with the code as a specification of the intended architecture in the future we integrated the architecture conformance analysis into the nightly build of the projects. The architecture description was inserted into the version control system so that the developers could access it easily. The results of the analysis can be accessed by the developers via a link in the project dashboard. The architecture got a more central role within the projects. The continuously checked architecture documentation in machine-readable form had a better availability and visibility within the projects than the text documents. All of the developers agreed that due to the integration of the architecture knowledge with the system using a continuous analysis of the consistency between the architecture description and the system, a better way of conserving the architectural knowledge within the projects could be achieved.

Furthermore, the continuous assessment enables a detection and resolution of potential architecture violations and design modifications early to the point in time when they were introduced. Thus, it can be decided very soon whether it is a violation or a design drift. So the costs of removing violations or adapting the documentation are rather low because the responsible developer still knows what he was working on and only low further efforts (like testing) have to be redone due to the modifications in the program.

Even several weeks after the main case study was done the developers reported that they looked at the results of the analysis in their project dashboards every morning. Due to the integration of the analysis into the nightly build an active way of managing the architecture knowledge within the projects was achieved. The architecture description in XML form stayed up-to-date (synchronized with the code) at least for the time we stayed in contact with the project (which was about a year in case of Project A).

Efforts needed. The efforts of establishing the analysis, the configuration of the dashboard and the creation of the XML architecture description cost about 5 days of work for each project. The most efforts were the reverse engineering and the discussions about how the appropriate architecture should look like. After the initial costs the nightly analyzes cause only little efforts to modify the architecture definition due to changes in the system. In Project A only four architectural changes happened in about a year. But this may of course vary between different project.

5 Threats to validity

Internal validity. Our evaluation of loss of architectural knowledge is based on several assumptions that have a potential to influence our results.

Hidden knowledge. Our work for identifying the intended architecture of the three systems is based on iterative inspections of the difference between the documented architecture and the code. These differences represent the basis of our discussions with the developers. However, it could happen that the documentation and the code match well in a certain respect (even if they are biased from the intended architecture). In these situations our method does not identify the intended architecture. Such situations influence the completeness of our approach and do not influence the quantitative results of the architectural loss (the loss of architectural knowledge would be even bigger in such cases).

Translation from textual documentation into checkable form. The translation of the informal documentation to the machine-readable XML-representation of the architecture might affect some of the results measured. Some information contained in the informal documentation might have been overlooked or misinterpreted, then the results measured after the initial execution of the analysis (step 2) might be biased. However, the same effect might take place when a developer that is unfamiliar with the system tries to learn the architecture by studying the documentation.

The developers themselves do not know exactly what the architecture was intended to look like. In some situations we asked the developers and the architect about some details of the architecture and they were not able to answer our questions immediately. There have been some questions that they had to discuss within the team before they could give us precise information about the architecture-to-be. This effect may influence the results of the case study because such a team decision might not reflect the intended architecture.

Hidden dependencies. There can be dependencies at the code level, e.g. generated by the use of reflection, that we did not analyze. In these cases the measurements would be flawed. However, our manual inspection of the code and the feedback of the developers made us be confident that we considered most common types of dependencies (as shown in Table 1).

External validity. There are several particularities of the investigated projects that could influence the generalization of our results to other projects.

The environment at MunichRE might influence the results. Subject to the case study have been three projects that have been carried out by different developers that are employed at different companies. Nevertheless, each of the project was done in the environment of MunichRE. All the projects used the same development process, a similar infrastructure and similar technologies (.NET). Due to that the external validity of the results may be limited. If the case study would have been conducted in another company the measures might vary. Additionally, three projects might be too few to gain representative results.

6 Related work

Checking the conformance of architecture. In the reverse engineering literature are several approaches for checking the architectural conformance with respect to high-level models [9, 4]. [4] proposes an approach to check the compliance of OO designs with the source code by mapping designs expressed in OMT to C++ programs. Our technology for specifying architecture and checking its conformance with the code is similar to the reflexion models developed by Murphy [9]. Even if our technique for checking the conformity of the architecture documentation with the code is similar, our focus is different. In this paper we investigated the loss of architectural knowledge in the systems evolution and the usefulness of explicit high-level models to make this knowledge explicit.

[3] proposes an approach to use declarative queries to specify structural dependencies and check them against the implementation. These queries are continuously run during the development. Our experience confirms the need to integrate architectural checks in the development process and thereby to prevent the loss of architectural knowledge.

Related case studies. [14] investigates how companies identify design erosion and address the preservation of the design. The case study is based on a qualitative analysis by performing interviews with developers. Among the major causes for the design erosion the authors identify the lack of knowledge of developers about the original design decisions and too little attention to design during evolution due to the pressure of releases. Our observations support these conclusions. We advocate that the preservation of architectural knowledge can be supported through continuous checking of the code and the documentation. Our approach in this paper is both quantitative (by measuring the discrepancies between the documentation and the code) and qualitative due to the contact with developers.

The case study in [12] illustrates the difficulty involved in detecting deviations of the code from the intended design that occur in the presence of personnel fluctuation. The authors propose the usage of metrics as principal means to detect the deviations. In this paper we advocate on the need to continuously maintain and check the consistency of an explicit representation of the intended architecture.

In [8] the authors present experiences from using architectural models in an industrial project. They report on huge efforts on keeping the models consistent with the implementation. Our results confirm the results obtained in this case study. Our approach of checking the consistency between the documentation (the model) and the implementation has the potential to reduce some of these efforts.

Architectural knowledge management. The authors of [1] draw a distinction between a *personalization strategy* and *codification strategy* for architectural knowledge management. On the one hand, the personalization strategy is mainly used in industry and emphasizes on the interaction among developers. On the other hand, the codification strategy is the basis for most of the research approaches in the area of knowledge management and concentrates on identifying and storing knowledge in artifacts and repositories [6]. In our work we advocate on the usefulness of the codification based approaches since they make the knowledge explicit, and this is of capital importance especially in the presence of personnel fluctuations when a pure personalization strategy is impossible. Furthermore, whenever differences are found by the analysis, they are catalizers for discussions among developers and entry points for a *personalization strategy*.

7 Conclusions

In this paper we present our experience in evaluating the loss of architectural knowledge in three industrial projects at MunichRE. Following our study we identified three manifestations of loss of architectural knowledge: decay of the code in form of violations of the intended architecture, loss of information in the documentation and different perceptions of the intended architecture by different developers. The central outcome of this case study is that we discovered that between 9% and 19% of all the dependencies implemented in the systems did not conform to the documented architecture. These differences could be identified as insufficiencies in the documentation as well as violations in the implemented architecture. So the real intended architecture was buried as implicit knowledge somewhere in between these artifacts as well as the knowledge of the developers and the architect. The main lesson learnt is that in order to minimize the knowledge loss we need to make the knowledge about the intended architecture explicit and perform automatic architecture conformance analyzes continuously in order to keep the awareness of developers about the architectural knowledge.

Acknowledgments We want to thank Benjamin Hummel and Florian Deißenböck for their efforts spent on the development of ConQAT [2] and the implementation of the architecture analysis. Furthermore, we want to thank our partners at MunichRE and the developers in the projects for the great collaboration.

References

[1] M. A. Babar, R. C. de Boer, T. Dingsoyr, and R. Farenhorst. Architectural knowlege management strategies: Approaches in research and industry. In *SHARK-ADI*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[2] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[3] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE*, pages 391–400, New York, NY, USA, 2008. ACM.

[4] R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *ICSM '98*, page 94, Washington, DC, USA, 1998. IEEE CS.

[5] D. Garlan. Software architecture: a roadmap. In *ICSE*, pages 91–101, New York, NY, USA, 2000. ACM.

[6] M. T. Hansen, N. Nohria, and T. Tierney. What's your strategy for managing knowledge? *Harvard Business Review 77 (2)*, pages 106–16, 1999.

[7] E. Koutsofios and S. North. Drawing graphs with dot, 1993.

[8] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. Experiences from representing software architecture in a large industrial project using model driven development. In *SHARK-ADI*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[9] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.

[10] S. Ornburn and S. Rugaber. Reverse engineering: resolving conflicts between expected and actual software designs. *CSM'92*, pages 32–40, Nov 1992.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[12] R. Tvedt, P. Costa, and M. Lindvall. Does the code match the design? a process for architecture evaluation. In *ICSM '02*, page 393, Washington, DC, USA, 2002. IEEE CS.

[13] J. van Gurp and J. Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[14] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17(4):277–306, 2005.